Protocol

App description:

Presentation Layer:
Packages: View, ViewModel...
Contents/Functionality: Responsible for the user interface, interaction, and visual representation of the application.
View classes (e.g., AddTourLogView, AddTourView, MainView) represent the different screens/views of the application.
ViewModel classes (e.g., AddTourLogViewModel, MainViewModel) handle the logic and data binding between the views and the underlying data/services.

Business Logic Layer:
Package: Service
Contents/Functionality: Implements the core business logic and provides services to other components.
Service classes (e.g., MapQuestRouteService, PDFService, RouteService) encapsulate specific functionalities and operations related to their respective domains.
These classes interact with the repositories and other services to perform the required operations.

Data Access Layer:
Package: Repository
Contents/Functionality: Handles data persistence and retrieval operations.
Repository classes (e.g., TourRepository, TourLogRepository) provide methods to interact with the underlying data storage (e.g., database) and perform CRUD (Create, Read, Update, Delete) operations on the entities.

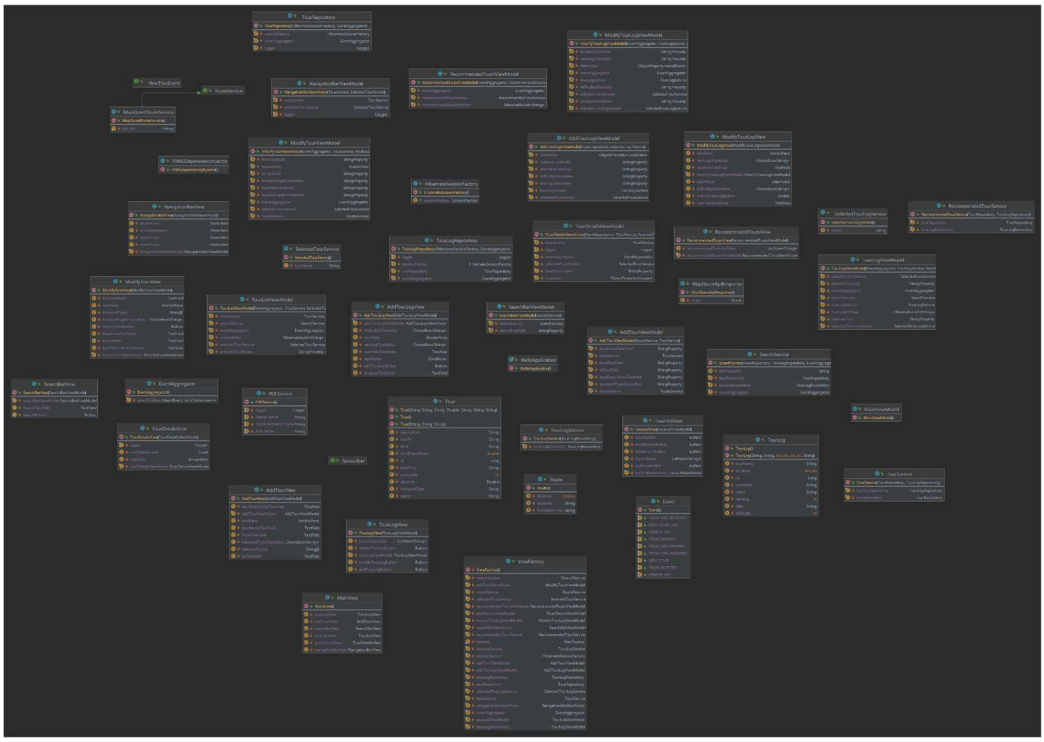
Model Layer:
Package: Model
Contents/Functionality: Represents the data entities used in the application.
Model classes (e.g., Tour, TourLog) represent the core entities with their properties and relationships.

Event Layer:
Package: Event
Contents/Functionality: Implements the event system to facilitate communication between different components.
Event classes (e.g., Event, NewTourEvent) define the different events that can occur in the application. The Event enum represents the various types of events that can be published.

Data Layer:
Package: Data
Contents/Functionality: Provides data-related utilities and resources. HibernateSessionFactory class manages the Hibernate session factory for data persistence.

Additional Components:
Resources: Contains FXML files for defining the application's user interface.
Test Folder: Contains unit tests for testing the functionality of the application's components.

Class diagram:
The class diagram is implemented as a puml file in the folder Diagram. Here are the classes:



Architectural Pattern: Model-View-ViewModel (MVVM)

The project follows the Model-View-ViewModel (MVVM) architectural pattern, which is commonly used in modern user interface (UI) development, particularly in frameworks JavaFX. MVVM is an extension of the Model-View-Controller (MVC) pattern that emphasises the separation of concerns and promotes testability and maintainability.

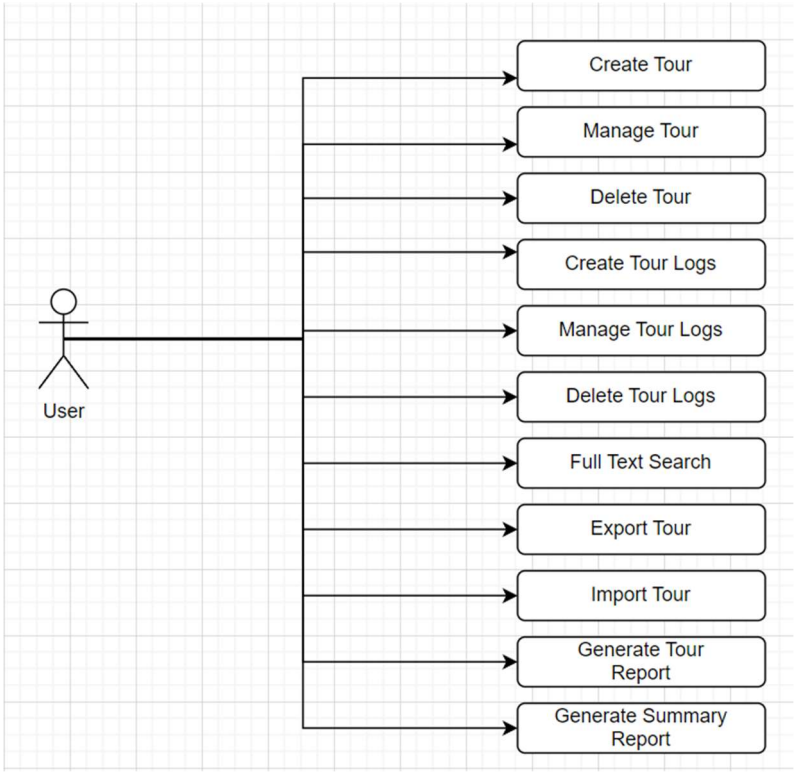
Key Components:

Model:
The Model represents the data and business logic of the application. It consists of classes such as Tour and TourLog, which encapsulate the relevant data and provide methods for data manipulation and storage. Additionally, repository classes like TourRepository and TourLogRepository handle data retrieval and persistence operations.

View:
The View corresponds to the UI components that are visible to the user. In this project, View classes include AddTourView, TourDetailsView, and other classes responsible for presenting the data and capturing user input. These Views are typically implemented using FXML files, defining the visual layout and appearance of the UI.

ViewModel:
The ViewModel acts as a mediator between the Model and the View. It provides the necessary data and behavior for the View to display and interact with. ViewModel classes, such as AddTourViewModel, TourDetailsViewModel, and others, encapsulate the presentation logic and facilitate data binding between the View and the underlying Model. They expose properties and commands that the View binds to, ensuring synchronization between the UI and the data.

Application features using an UML use case diagram



Library decisions

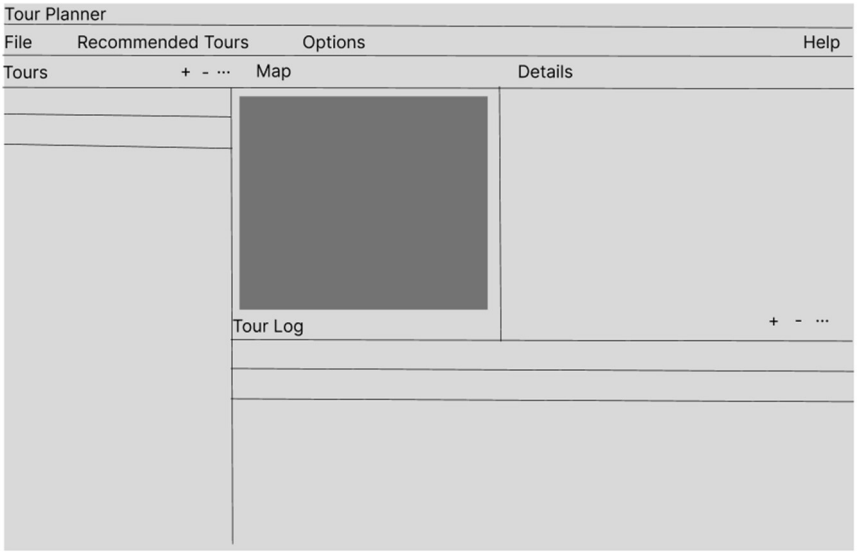
Log4j:
Log4j was chosen as the logging framework for the project. It provides a reliable and flexible logging mechanism, allowing for comprehensive monitoring and troubleshooting of the application.
MapQuest:
The MapQuest library was incorporated to enhance the application's routing and mapping capabilities. By leveraging the MapQuest API, the project can retrieve accurate geolocation data, calculate distances, and display interactive maps.
iTextPDF:
The iTextPDF library was selected to handle PDF generation and manipulation within the application. iTextPDF offers extensive functionality for creating, editing, and rendering PDF documents.
Hibernate:
Hibernate was chosen as the object-relational mapping (ORM) framework for the project. It simplifies data persistence by providing an abstraction layer between the application and the underlying database.

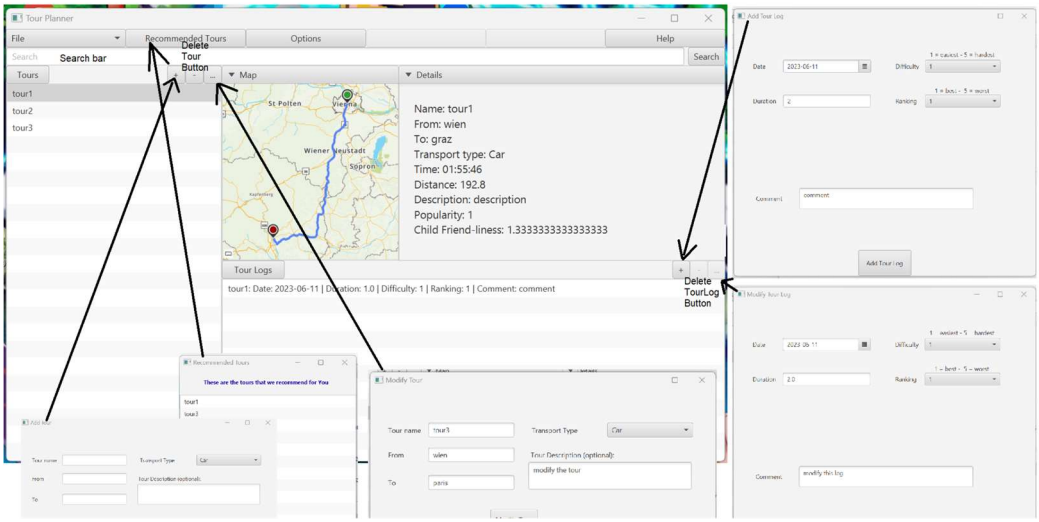
lessons learned
Throughout the development of the project, several valuable lessons were learned:
MVVM: Adhering to the principles of separation of concerns and modular architecture is essential for code maintainability and scalability. Clear separation between layers (such as Model, View, and ViewModel) facilitates easier code management.
Logger: Implementing robust error handling mechanisms and integrating a logging framework, such as Log4j, greatly enhances the project's stability and helps identify and resolve issues promptly.

Design pattern
The Observer pattern is implemented through the EventAggregator and Event classes in the Event layer. The EventAggregator acts as the central hub for event publishing and subscription. Other components, such as the ViewModel classes, subscribe to specific events using the addSubscriber() method. When an event is published, the EventAggregator notifies all the subscribed components, allowing them to react and perform the necessary actions. This decoupling of components through event-based communication follows the Observer design pattern, where the observers (subscribers) are notified of changes in the state of the subject (event).

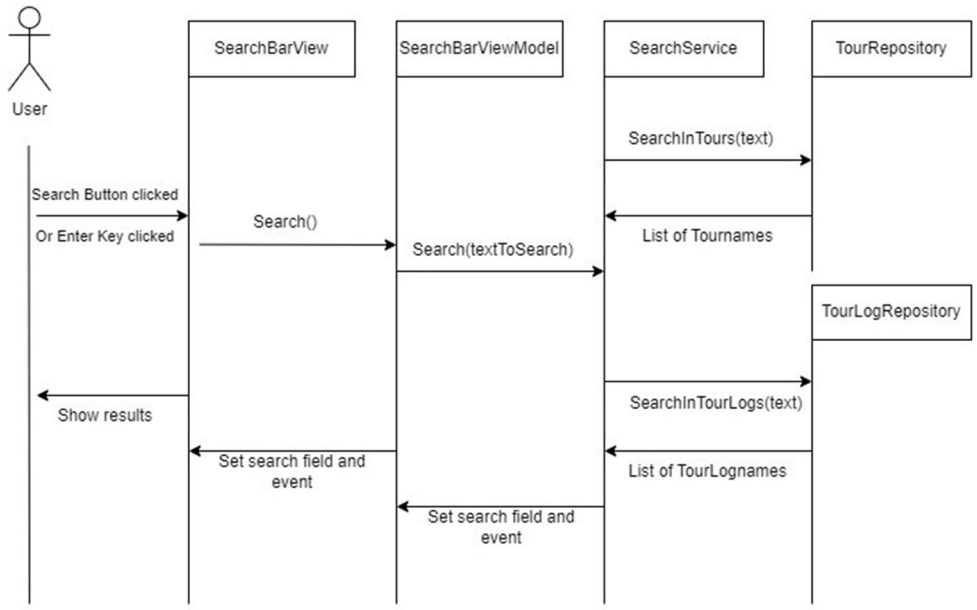
UI/UX - wireframes

The user interface of the app is designed to provide a seamless and intuitive experience for users. With easy-to-use features like search bars, tour lists, and recommended tours, users can quickly find and explore various tours, enhancing their overall journey planning experience.





Sequence diagram for full-text search



Use Case: Perform Full-Text Search
Actor: User
Description: The user wants to perform a full-text search.
Steps:
The user enters the search text in the search bar.
The user clicks the search button or presses the enter key.
The system initiates the search process.
The system retrieves the search results from the SearchService.
The system displays the search results in the UI.

Unique Feature: Recommended Tours

The unique feature of the application is the "Recommended Tours" functionality. This feature is powered by the "RecommendedTourService" class, which identifies and presents a curated list of tours that are highly recommended based on their popularity and average ranking provided by users. The "RecommendedTourService" class communicates with the "TourRepository" and "TourLogRepository" to fetch the required data. The "RecommendedToursViewModel" class manages the recommended tours list and interacts with the event system. Finally, the "RecommendedToursView" class is responsible for displaying the recommended tours to the users. With this feature, users can easily discover and explore the tours that come highly recommended, considering factors like popularity and user rankings.
A tour can be recommended only if it has a popularity of at least 1 (measured by the number of tour logs) and an average ranking that does not exceed 2.5.

Tracked Time:

Understanding the project requirements - 5 Hours
Creating the Tour and TourLog models - 4 Hours
Implementing the TourRepository and TourLogRepository - 4 Hours
Setting up the Hibernate integration (HibernateSessionFactory) - 4 Hours
Designing the layout, adding UI elements, fxml files - 8 Hours
Implementing services such as MapQuestRouteService, PDFService, RouteService, RecommendedTourService, SearchService, etc. - 16 Hours
Integrating third-party libraries like log4j - 4 Hours
Creating the necessary views e.g. AddTourView - 8 Hours
Implementing the corresponding view models e.g. AddTourViewModel - 8 Hours
Handling UI events, data binding - 8 Hours
Implementing the event system - 4 Hours
Testing and bug fixing - 8 Hours
Documentation - 4 hours

Total: 83 hours

Unit Test decisions

We have selected 21 unit tests to verify the crucial components of our code and ensure their proper functionality. These tests are essential as they cover critical functionalities that are vital for the correct operation of our application. By executing these tests, we can identify any issues or bugs at an early stage and ensure that our application functions as intended. Here are the most significant tests:

+ We test that the program does not crash due to false user input.
+ We test whether the fields in the windows are initialized correctly as per our requirements.
+ We test whether the fields are appropriately set when the function is called.
+ For example, in the AddTourViewModelTest, we verify that the method 'saveTour' is called correctly.
+ We test whether events are published as intended.
+ We verify whether the list of tours displays the expected content.

Git Link: https://github.com/RidhaOm/TourPlanner